A Review of Approaches to Detecting Software Design Patterns

Jameleh Asaad, Elena Avksentieva
ITMO University
St. Petersburg, Russia
jamelehasaad@gmail.com, eavksenteva@itmo.ru

***Abstract*—Design patterns play a crucial role in modern software engineering, providing reusable solutions to common design challenges. Among the most influential collections of design patterns is the Gang of Four (GoF) patterns, which offer a timeless framework for addressing recurring design problems. This article investigates the enduring impact of GoF design patterns on software development practices, examining their utilization in contemporary software projects and frameworks. Additionally, this study conducts a thorough analysis of various design pattern detection approaches, evaluating their effectiveness and implications in real-world software development contexts. By combining theoretical frameworks with empirical studies, we aim to provide valuable insights into the role of design patterns in software engineering and offer guidance on selecting appropriate detection methods for software project.**

I. Introduction

In the vast landscape of software engineering, the integration of design patterns stands as a foundational pillar, guiding developers towards the creation of robust, maintainable, and scalable software solutions. Originating from architectural principles and later adapted to the realm of software development, design patterns have evolved into indispensable tools, catalyzing significant advancements in the field. Since their inception, these patterns, epitomized by the seminal work of the Gang of Four (GoF), have permeated the fabric of software design, offering standardized solutions to recurring design challenges.

The enduring legacy of design patterns is perhaps best exemplified by the seminal work ”Design Patterns: Elements of Reusable Object-Oriented Software,” authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This monumental text, commonly referred to as the GoF book, serves as a compendium of time-tested solutions to common software design problems, providing developers with a shared vocabulary and framework for effective collaboration.

Amidst the proliferation of design patterns across various domains of software development, a critical question emerges: What tangible impact do these patterns have on software quality? This question serves as the crux of our inquiry, as we embark on a comprehensive exploration of the influence of GoF design patterns on software quality metrics and outcomes.

Furthermore, the field of design pattern detection (DPD) has emerged as a pivotal area of research, essential for supporting software maintenance and reverse engineering efforts. By providing valuable insights into the structure and organization of complex software systems, DPD facilitates comprehension and subsequent reengineering steps. However, achieving optimal design pattern detection remains a challenge due to factors such as differing pattern definitions and subjective interpretation.

Integral to our exploration is the examination of popular frameworks that extensively employ GoF design patterns, along with their associated detection methods. By dissecting how these frameworks leverage design patterns to achieve modularity, extensibility, and maintainability, and exploring the methodologies used to detect them, we aim to provide a nuanced understanding of the practical implications of design pattern utilization on software quality.

Through empirical studies, case analyses, and industry best practices, we endeavor to unravel the intricate relationship between the judicious application of GoF design patterns, their detection, and various dimensions of software quality. Ultimately, this research aspires to offer actionable insights for software practitioners and decision-makers, guiding them in making informed choices regarding the incorporation of design patterns into their software projects.

In the subsequent sections, we will traverse the landscape of software quality assessment, elucidating key dimensions and metrics, before embarking on a systematic examination of the impact of GoF design patterns and their detection methods on each facet of software quality. By synthesizing insights from theory, practice, and empirical research, we aim to contribute to the ongoing dialogue surrounding the role of design patterns in engineering software systems of enduring excellence.

II. Background and related works

A. Introduction to software design patterns

Software design patterns, derived from architectural principles, have evolved as fundamental solutions to recurring problems in software engineering. Initially conceptualized by Alexander et al. within the realm of architecture, these patterns were later adapted to software development in 1987, catalyzing a significant advancement in the field [1], [2]. Since then, software design patterns have become indispensable tools for improving software quality and fostering collaboration among developers [3].

Today, a multitude of design patterns exists, spanning various levels of abstraction within software systems. These patterns encompass architectural, design, and implementation aspects, addressing a wide array of domain-specific challenges [4], [5], [6], [7]. Notably, seminal works such as "Design Patterns: Elements of Reusable Object-Oriented Software" by the Gang of Four have revolutionized software design, providing a standardized framework for addressing common design problems [8], [9].

B. Categorization of design patterns

Design patterns are typically categorized into three main classes:

1) Structural Design Patterns: These patterns focus on organizing objects and classes to form larger, functional structures. Examples include Adapter, Decorator, and Composite patterns [10]
2) Creational Design Patterns: Primarily concerned with object creation mechanisms, these patterns encapsulate the instantiation process, promoting flexibility and scalability. Examples include Factory Method and Singleton patterns [11].
3) Behavioral Design Patterns: These patterns define communication patterns among objects, emphasizing the distribution of responsibilities to enhance flexibility and maintainability. Examples include Observer, Strategy, and Command patterns [11], [12].

C. Challenges in implementing design patterns

Despite their benefits, developers often encounter challenges when implementing design patterns [13]:

1) Pattern Selection: Choosing the appropriate design pattern amidst similarities and complexities can be daunting.
2) Understanding Pattern Characteristics: Grasping the nuances of unfamiliar patterns poses a significant learning curve for developers.
3) Refactoring: Incorporating design patterns into existing architectures requires careful consideration and execution.
4) Trade-offs of Quality Attributes: Balancing various quality attributes, such as maintainability and scalability, can be complex when implementing specific design patterns.
5) Technological Problems: Implementing design patterns may introduce technological challenges that require innovative solutions.

D. Significance of detecting design patterns

Detecting design patterns within codebases is crucial for effective software development, especially during maintenance and refactoring tasks. It serves as a fundamental step towards addressing many identified issues, including pattern selection, understanding pattern characteristics, and optimizing quality attribute trade-offs. By emphasizing the significance of detection, developers gain insights into the complexities and nuances involved in this process. Automated tools and algorithms complement manual inspection, enabling developers to leverage existing solutions and build maintainable software systems. As elucidated in the scholarly article, detecting design pattern instances from source codes in software re-engineering offers several benefits:

1) Understanding Complex Systems: By identifying design patterns in source code, developers can gain insights into the underlying structure and organization of complex software systems. This understanding is crucial for effective re-engineering efforts.
2) Improving Software Quality: Detecting design patterns allows developers to assess the quality of the software architecture. By recognizing well-known design patterns, they can ensure that the system follows established best practices and principles.
3) Facilitating Refactoring: Design patterns provide proven solutions to common design problems. Detecting these patterns can guide developers in refactoring the codebase to improve maintainability, extensibility, and overall quality.
4) Enhancing Program Understanding: Recognizing design patterns in source code enhances program comprehension. Developers can quickly grasp the intent and structure of the software by identifying familiar patterns.
5) Software Documentation: Design patterns serve as a form of documentation for software systems. Detecting these patterns helps in documenting the design decisions and architectural choices made during the development process.
6) Software Maintenance: Design pattern detection aids in software maintenance by enabling developers to identify areas of the codebase that adhere to specific design patterns. This knowledge simplifies the process of making changes and updates to the software.

In summary, understanding and detecting design patterns are critical components of software engineering, enabling developers to build robust, maintainable, and scalable software systems.

III. The research questions and methodology

A. Research Questions

In this study, we aim to delve into the impact of design patterns on the software development process, scrutinizing both their advantages and limitations, with a particular focus on the Gang of Four (GoF) design patterns in contemporary software development. To steer our investigation, we have articulated three primary research questions:

RQ1: Do Gang of Four (GoF) design patterns continue to hold significance in contemporary software development?

RQ2: What effect does the utilization of GoF design patterns have on software quality?

RQ3: What methodologies are employed in the detection of design patterns?

B. Methodology

To conduct a comprehensive examination of the current state of knowledge regarding software design patterns, we formulated meticulous research questions and undertook a systematic literature review. The primary objective was to gather empirical evidence meeting predefined inclusion criteria to effectively address our research inquiries and hypotheses. To ensure methodological rigor and minimize potential biases, we adopted explicit and structured methodologies throughout the review process, resembling a qualitative systematic review [14].

1) Formulating research questions: In the initial stage, we precisely defined the research problem and crafted structured research inquiries to effectively address the identified problem.
2) Literature search and selection: We conducted a meticulous examination of a corpus comprising over 40 scientific articles using reputable databases such as Scopus, Web of Science, PubMed, IEEE Xplore, ScienceDirect, and the Directory of Open Access Journals (DOAJ). Publications were systematically categorized based on thematic relevance.
3) Quality assessment: Both conference proceedings and journal articles underwent rigorous evaluation based on substantive content, impact factors, citation counts, and H-index values. Preference was given to articles with high citation counts, significant impact factors, and informative content.
4) Data analysis and interpretation: Upon acquiring necessary information, we conducted a comprehensive statistical analysis to derive meaningful interpretations and conclusions, synthesizing insights from each publication.

We established specific criteria for selecting articles for inclusion in our research, focusing on those explicitly addressing design patterns, particularly GoF design patterns, or involving the identification or classification of design patterns. After initial screening, shortlisted publications underwent further evaluation based on relevance to our research objectives, citation counts, and publication year.

The findings of this systematic approach are presented in the Results and Analysis section.

IV. Results and analysis

A. GoF design patterns use cases in today perspectives

We present examples of how GoF design patterns are integrated into modern software projects and frameworks to enhance code quality, maintainability, and scalability:

1) Creational Patterns
   - Singleton Pattern: Many modern frameworks and libraries use the singleton pattern to ensure that only one instance of a class is created and provide a global point of access to that instance. For example, in Java, the Spring Framework uses singleton beans for managing application components [15], [16], [17].
   - Factory Method Pattern: Frameworks like Django (Python web framework) use factory methods to create instances of model objects, providing a centralized way to create different types of objects without exposing their instantiation logic [18], [19].
2) Structural Patterns
   - Adapter Pattern: Libraries like Retrofit (for Android) use the adapter pattern to convert the interface of a class into another interface that a client expects. This allows seamless integration with existing systems [20].
   - Decorator Pattern: GUI toolkits such as Swing (Java) use the decorator pattern to add behavior or responsibilities to objects dynamically. For instance, adding borders or scrollbars to components [21].
3) Behavioral Patterns
   - Observer Pattern: Modern UI frameworks like React (JavaScript) use the observer pattern extensively for managing state and updating UI components reactively based on changes in data [22].
   - Strategy Pattern: Frameworks such as TensorFlow (for machine learning) utilize the strategy pattern to encapsulate algorithms and make them interchangeable. This allows users to switch between different training or optimization strategies easily [23].

Additionally, the use of design patterns is not limited to frameworks or libraries, they can be applied in various domains.For instance, the Abstract Factory pattern aids in managing the creation of GUI components and structures by abstracting specific implementations from the client code. This abstraction layer provided by the Abstract Factory pattern simplifies the integration process of different GUI technologies and enables a more modular and flexible design approach [24].

B. GoF design patterns impact on software quality

One of the primary concerns for developers employing design patterns in application development is the impact on quality attributes. Determining how design patterns influence software quality poses a significant challenge. Previous research has utilized surveys, experiments, case studies, and analytical techniques to assess the use of design patterns. However, analyzing pattern interactions or coupling complicates the assessment of their impact on quality attributes [13]. Referring to the ISO/IEC 9126 standard as a framework, six primary quality attributes are identified: portability, usability, efficiency, maintainability, functionality, and reliability [25]. These attributes are further delineated into various subcharacteristics, as illustrated at Fig. 1.

Among the 23 Gang of Four (GoF) design patterns, observer, state, factory method, composite, singleton, decorator, and strategy are extensively evaluated for their impact on software quality [13]. Design patterns, regarded as best practice solutions, encapsulate collective design experience and embody fundamental principles of object-oriented design [26]. Novices benefit from design patterns as shortcuts to producing

Fig. 1. Quality Attributes of ISO 9126 [25]

high-quality solutions, significantly accelerating their learning curve [8]. Moreover, incorporating design patterns into quality evaluation models underscores their significance in ensuring software quality [27]. This designation as best practices stems from their adherence to core design principles, such as encapsulation and the open-closed principle, which mitigate bugs introduced through code changes [28]. Furthermore, principles like ”program to an interface not an implementation” and ”favor object composition over class inheritance” reduce class dependencies and promote a focused hierarchy of classes.

However, while different design patterns implement diverse sets of design principles to address various challenges, their effectiveness in enhancing different quality aspects varies [29]. Among these aspects, maintainability emerges as the most crucial, according to respondents [30]. Nevertheless, studies present conflicting views on the impact of design patterns on maintainability. Some suggest that while design patterns are generally advisable, they may not always be optimal, and exercising common sense in their application is recommended, especially in cases of uncertainty [31]. Additionally, different patterns have varied impacts on maintainability; for instance, while patterns like Observer and Decorator are easily understood, Composite poses challenges due to recursion [32]. Conflicting results from studies highlight the complexity of this issue, with some finding non-pattern-based versions more maintainable, while others report no discernible impact of design patterns on maintainability and understandability [33], [34], [35].

Certain studies focus on specific design patterns' impact; for example, one study found that using patterns like State, Composite, and Chain of Responsibility made diagrams harder to understand and modify [36]. Conversely, another study concluded that while the Visitor pattern requires more time for comprehension and modification, its canonical form reduces the effort for modification tasks [37]. Furthermore, a study reported an improvement in maintainability with increased use of design patterns, particularly evident in the JHotDraw software system [38].

C. Design pattern detection methods

1) Similarity scoring approach: Some papers have employed the Similarity Scoring Approach (SSA) for detecting design patterns using matching algorithms. For this purpose, researchers developed a Java-based tool geared towards implementing a similarity algorithm, likely intended for data analysis or comparison tasks. To evaluate the algorithm's performance, metrics such as False Positive (FP), False True (FT), and recall were considered, commonly used in machine learning and data analysis. Furthermore, the tool's impact on CPU and memory utilization was assessed, providing insights into its performance. System representation relied on a UML Class diagram, a standard notation for visualizing system design, with the diagram converted into a binary matrix to encode relationships computationally. Distinguishing between different relationships in UML diagrams typically involves utilizing various symbols and annotations [39].

Regarding threats to validity and limitations, reliance on manual code inspection for identifying pattern instances poses a risk of introducing false negatives. Additionally, the approach may struggle to detect patterns based on specific action sequences since it does not incorporate dynamic information, although it could be complemented by approaches using dynamic data. Scalability of the methodology is hindered by the time required for the similarity algorithm to converge, especially with larger and denser subsystem matrices, leading to increased memory requirements. Inserting novel design patterns into the tool is relatively straightforward if their characteristics align with existing attribute matrices, but introducing new characteristics requires additional implementation effort. As the number of supported patterns increases, existing attribute matrices are expected to become more adept at describing various structural characteristics.

2) MARPLE-DPD [40]: Metrics and Architecture Reconstruction Plugin for Eclipse, is a tool implementing DPD (Design Pattern Detection) from Java source code alongside additional functionalities. The architecture of MARPLE, as illustrated at Fig. 2, involves three main modules as: the Information Detector Engine, the Joiner, and the Classifier. The Information Detector Engine builds the system model

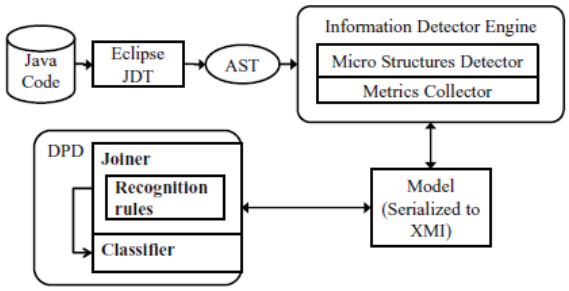

Fig. 2. The architecture of MARPLE [40]

and collects micro-structures and metrics from the abstract syntax trees (ASTs) of the analyzed system. The Joiner extracts potential design pattern candidates based on micro-structures, while the Classifier evaluates whether the groups of classes detected by the Joiner represent realizations of design patterns. The Micro-structures detector extracts micro-structures using visitors parsing an AST representation of the source code. Experimental investigations focused on the accuracy achieved by different machine learning models, with

a dataset comprising ten projects. The evaluation of design pattern instances involved considering various criteria to avoid conflicting evaluations. The Joiner analyzes information from the MSD to extract groups of classes as pattern candidates, while the Classifier assesses the similarity of pattern instances to previously classified design patterns. The detection process of MARPLE-DPD comprises phases supported by different modules, including the extraction of design pattern candidates, matching and merging steps, and pattern instance representation. Experiments on MARPLE-DPD were conducted using datasets representing commonly used design patterns, with results obtained through various machine learning models.

3) Graph-based approach: The researchers addressed the problem of identifying design patterns in software source code and proposed a five-step method to solve it [41]. Initially, the system's structure was extracted from the source code and represented as a class diagram. Then, directed semantic graphs were constructed to represent the system and specified patterns, focusing on relationships between classes. Pattern graphs were built based on the main structure of each pattern, and the graphs were enriched to include indirect relationships between classes. A matching algorithm was used to find pattern instances in subsystem graphs, and the behavioral signature of patterns was analyzed to eliminate false positives. The method was evaluated using three open-source projects, comparing results with existing tools. Precision and recall metrics were used to evaluate the effectiveness of the method, showing promising results. Threats to validity were discussed, including potential inaccuracies in measurements and generalization to other systems. The paper concluded by highlighting the importance of design pattern detection in software engineering and outlining future research directions, including standardizing pattern signatures and extending semantic analysis of design motifs.

Another proposed method focuses on design pattern detection using a greedy algorithm based on inexact graph matching [42]. The algorithm decomposes the graph matching process into phases, where the value of K ranges from 1 to the minimum of the numbers of nodes in the two graphs to be matched. By using small values of K, the algorithm significantly reduces the search space while still producing very good matchings between graphs. This approach aims to provide an automatic and reliable way to discover design patterns in system designs, which can enhance program understanding and software maintenance. The algorithm compares the nodes and edges of two graphs to find the matching that leads to the minimum matching error, defined as the dissimilarity between matched nodes and corresponding edges. This error represents the distance between the two graphs, allowing for effective pattern detection even in the presence of noise and distortion.

4) Probabilistic Approach: The method proposed for probabilistic detection of Gang of Four (GoF) design patterns in source code consists of two main phases: the Learning phase (Phase I) and the Detection phase (Phase II) [43].

In the Learning phase, a trained Multilayer Perceptron (MLP) model is created by identifying relevant features extracted from the source code. These features are determined based on the concept of design pattern signatures, representing patterns as a set of features. The source code is then used to train the MLP model, which will subsequently be utilized in the Detection phase for probabilistic detection of design patterns.

During the Detection phase, the trained model is applied to the source code to detect design patterns probabilistically. The source code is first converted into a class diagram, which is then transformed into an enriched graph. Candidate instances of design patterns are extracted from this graph based on connected nodes. The method employs regression analysis to differentiate between design patterns and considers the set of features representing patterns.

Overall, this proposed method utilizes a probabilistic approach to detect design patterns in source code, with the aim of enhancing coverage and distinguishing between design pattern variants. By leveraging machine learning techniques and feature representation, the method strives to improve the accuracy and effectiveness of design pattern detection in software systems.

5) features-based approach: In this scholarly approach, design patterns are scrutinized as aggregations of features, delineating aspects such as intent, structure, behavior, and sample code. A novel pattern specification technique is introduced with the precise aim of facilitating automated application while ensuring comprehensibility and adaptability for human users. This method identifies recurrent sub-structures, termed features, among diverse pattern variations, encompassing elements such as classes, relationships, and method return types.

In detailing their research methodology, the scholars present a meticulously structured two-stage pattern recognition process [44]. In the initial stage, they meticulously formulate semi-formal pattern definitions based on prevalent extensible feature types, derived from exhaustive analyses of GoF patterns and their myriad variations. Subsequently, the second stage entails pattern identification through the discernment of their defining features using an array of sophisticated search technologies.

Stage 1 involves defining patterns using a catalog of feature types and hierarchical pattern definitions, including pattern catalogs, pattern definitions, variant definitions, and features. Feature types are reusable elements representing elementary and recurring features across patterns, while pattern definitions consist of features organized into variants.

Stage 2 entails recognizing defined patterns by iterating through each feature of each variant definition, selecting appropriate search technologies, and executing queries. Techniques include repository queries, specific parser modules, and regular expressions. The process iteratively prunes or extends candidate patterns based on search results.

An illustrative example demonstrates the recognition procedure, where a Factory Method pattern is detected within source code based on the defined features. The process involves sequentially searching for features and evaluating candidate patterns based on search results, resulting in the identification of valid pattern instances.

Another approach combines feature extraction, machine learning classification, and thorough evaluation to automate the identification of design patterns in Java source code [45]. The methodology involves the creation of a new corpus (DPDF-Corpus) sourced from GitHub, focusing exclusively on design patterns. Feature extraction encompasses 15 selected features based on syntactic and semantic constructs of Java, capturing both class-level and method-level information. The design pattern detection process involves preprocessing, where structural, syntactic, and linguistic representation (SSLR) is generated using call graphs and code parsing. Model building employs the Word2Vec algorithm to create a Java Embedded Model (JEM) from SSLR. Supervised classifiers, based on randomized decision trees, are trained on the labelled corpus to predict design pattern instances. The corpus is labelled by annotators with expertise in Java programming and design patterns. Machine classification utilizes ensemble learning with randomized decision trees, and evaluation is conducted using standard metrics such as Precision, Recall, and F1-Score, demonstrating the effectiveness of the approach in automated detection of software design patterns in Java source code.

6) A grammar-based evolutionary machine learning approach: [46] The GEML approach is a two-phased model for detecting design patterns (DPs) in source code.In the first phase, the system learns structural, behavioral, and metric-based properties of DPs by analyzing a repository containing labeled DP implementations. It employs the G3P4DPD algorithm to generate class association rules (CARs) compliant with a Context-Free Grammar (CFG) formalizing the syntax of DPs. These rules are then pruned and arranged to form the detection model. The second phase involves applying the G3P4DPD algorithm iteratively to refine the rules and construct an external archive of the most accurate rules.The overall structure of GEML is depicted in Fig. 3.

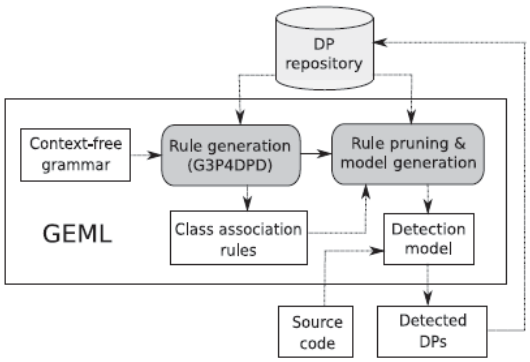

Fig. 3. The GEML approach [46]

This algorithm employs genetic operators such as crossover and mutation to evolve the rules based on their support in the code repository. Each individual rule's genotype is represented as a valid derivation tree according to the CFG, while its phenotype denotes the corresponding class association rule. This process ensures that the detection model adapts to the specific development culture over time, enhancing its effectiveness in identifying DPs in source code.

V. Conclusion

In conclusion, this study sheds light on the enduring impact of Gang of Four (GoF) design patterns on the software development landscape. Through an exploration of their utilization in modern software projects and frameworks, we have demonstrated the significant role that design patterns play in enhancing code quality, maintainability, and scalability. Despite the challenges and conflicting findings regarding their impact on software quality, design patterns remain invaluable tools for software practitioners seeking to engineer high-quality software systems.

As we navigate the complexities of contemporary software development, understanding the implications of design pattern selection on software quality is paramount. By leveraging the insights gleaned from our comprehensive analysis, software practitioners can make informed decisions and adopt best practices to ensure the successful integration of GoF design patterns into their software projects, thereby advancing the state of software engineering excellence.

The exploration of popular frameworks that extensively employ GoF design patterns has provided valuable insights into how these patterns contribute to modularity, extensibility, and maintainability in real-world software development. By dissecting the practical implications of design pattern utilization on software quality, we have highlighted the importance of strategic pattern selection in software projects.

In essence, the legacy of the Gang of Four design patterns endures as a foundational pillar in software engineering, offering a timeless framework for addressing recurring design challenges. As software practitioners continue to innovate and evolve, the judicious application of design patterns, particularly those advocated by the GoF, remains a cornerstone for achieving software quality and sustainability in a rapidly changing technological landscape. In addition to exploring the utilization of GoF design patterns in modern software projects and frameworks, this study also conducted a thorough analysis of various design pattern detection approaches. Leveraging insights from both theoretical frameworks and empirical studies, we systematically evaluated the effectiveness and implications of these detection methods in real-world software development contexts. By examining the strengths, limitations, and practical considerations associated with different detection techniques, we aimed to provide software practitioners with valuable guidance on selecting the most suitable approach for their projects. Through this comprehensive analysis, we have contributed to a deeper understanding of how design patterns are identified, validated, and integrated into software systems, thereby enhancing the quality, maintainability, and scalability of modern software applications.

References

[1] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Constructions*. Center for Environmental Structure

Berkeley, Calif: Center for Environmental Structure series. Oxford University Press, 1977.
[2] P. Kuchana, *Software Architecture Design Patterns in Java*. CRC Press, 2004.
[3] M. Di Penta, L. Cerulo, Y. G. Guéhéneuc, and G. Antoniol, "An empirical study of the relationships between design pattern roles and class change proneness," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 217–226.
[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. John Wiley & Sons, 1996.
[5] M. Fowler, *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1997.
[6] S. Bjork and J. Holopainen, *Patterns in Game Design (Game Development Series)*. Charles River Media, Inc., 2004.
[7] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.
[9] T. Taibi, *Design Pattern Formalization Techniques*. IGI Publishing, 2007.
[10] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem, "Object-oriented design patterns recovery," *Journal of Systems and Software*, vol. 59, no. 2, p. 181–196, 2001.
[11] J. L. Hodges, "Design patterns," in *Software Engineering from Scratch*. Springer, 2019, p. 293–304.
[12] M. Saeki, "Behavioral specification of gof design patterns with lotos," in *Proceedings Seventh Asia-Pacific Software Engineering Conference, APSEC 2000*. IEEE, 2000, p. 408–415.
[13] M. Rahman, M. S. H. Chy, and S. Saha, "A systematic review on software design patterns in today's perspective," in *2023 IEEE 11th International Conference on Serious Games and Applications for Health (SeGAH)*. IEEE, 2023, pp. 1–8.
[14] M. Dixon-Woods, R. Fitzpatrick, and K. Roberts, "Including qualitative research in systematic reviews: opportunities and problems," *Journal of evaluation in clinical practice*, vol. 7, no. 2, pp. 125–33, 2001.
[15] A. Lilleaas, "Setup, teardown, and dependency injection with spring context," in *Pro Kotlin Web Apps from Scratch: Building Production-Ready Web Apps Without a Framework*. Berkeley, CA: Apress, 2023, pp. 251–259.
[16] P. Späth and et al., "Introducing ioc and di in spring," in *Pro Spring 6 with Kotlin: An In-depth Guide to Using Kotlin APIs in Spring Framework 6*. Berkeley, CA: Apress, 2023, pp. 41–101.
[17] F. Fábry, "Java microservice migration to the spring framework," [Details about publication].
[18] D. Fr a szczak, "Nefbdaa—. net environment for building dynamic angular applications," *SoftwareX*, vol. 19, p. 101163, 2022.
[19] N. Denissov, "Creating an educational plugin to support online programming learning: A case of intellij idea plugin for a+ learning management system," 2021.
[20] M. Wilkes and M. Wilkes, "Alternative interfaces," in *Advanced Python Development: Using Powerful Language Features in Real-World Applications*, 2020, pp. 183–245.
[21] Z. Azimullah, Y. S. An, and P. Denny, "Evaluating an interactive tool for teaching design patterns," in *Proceedings of the Twenty-Second Australasian Computing Education Conference*, 2020, pp. 167–176.
[22] A. Osmani, *Learning JavaScript Design Patterns*. O'Reilly Media, Inc., 2023.
[23] F. J. Király, M. Löning, A. Blaom, A. Gueciouer, and R. Sonabend, "Designing machine learning toolboxes: Concepts, principles and patterns," *arXiv preprint arXiv:2101.04938*, 2021.
[24] A. Korunović and S. Vlajić, "An example of integration of java gui desktop technologies using the abstract factory pattern for education purposes," *ETF Journal of Electrical Engineering*, vol. 29, no. 1, pp. 3–11, 2023.
[25] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Research state of the art on gof design patterns: A mapping study," *Journal of Systems and Software*, vol. 86, no. 7, pp. 1945–1964, 2013.
[26] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005, p. 295–304.
[27] Z. Balanyi and R. Ferenc, "Mining design patterns from c++ source code," in *Proceedings of the International Conference on Software Maintenance*, 2003, p. 305–314.
[28] E. Freeman, B. Bates, K. Sierra, and E. Robson, *Head First Design Patterns*. O'Reilly Media, 2004.
[29] L. Prechelt, B. Unger, W. Tichy, P. Brossler, and L. Votta, "A controlled experiment in maintenance: comparing design patterns to simpler solutions," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, p. 1134–1144, 2001.
[30] B. Bontchev and E. Milanova, "On the usability of object-oriented design patterns for a better software quality," *Cybernetics and Information Technologies*, vol. 20, no. 4, pp. 36–54, 2020.
[31] L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta, "A controlled experiment in maintenance: comparing design patterns to simpler solutions," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1134–1144, 2001.
[32] M. Vokáč, T. Walter, L. K. S. Dag, A. Erik, and A. Magne, "A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment," *Empirical Software Engineering*, vol. 9, no. 3, pp. 149–195, 2004.
[33] L. Prechelt and M. Liesenberg, "Design patterns in software maintenance: An experiment replication at freie universität berlin," in *Second International Workshop on Replication in Empirical Software Engineering Research (RESER)*, 2011, pp. 1–6.
[34] N. Juristo and S. Vegas, "Design patterns in software maintenance: An experiment replication at upm - experiences with the reser'11 joint replication project," in *Second International Workshop on Replication in Empirical Software Engineering Research (RESER)*, 2011, pp. 7–14.
[35] A. Nanthaamornphong and J. C. Carver, "Design patterns in software maintenance: An experiment replication at university of alabama," in *Second International Workshop on Replication in Empirical Software Engineering Research (RESER)*, 2011, pp. 15–24.
[36] J. Garzás, G. Félix, and M. P., "Do rules and patterns affect design maintainability?" *Journal of Computer Science and Technology*, vol. 24, no. 2, pp. 262–272, 2009.
[37] S. Jeanmart, Y. G. Gueheneuc, H. Sahraoui, and N. Habra, "Impact of the visitor pattern on program comprehension and maintenance," *3rd International Symposium on Empirical Software Engineering and Measurement ESEM*, pp. 69–78, 2009.
[38] P. Hegedűs, B. Dénes, F. Rudolf, and G. Tibor, "Myth or reality? analyzing the effect of design patterns on software maintainability," pp. 138–145, 2012.
[39] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE transactions on software engineering*, vol. 32, no. 11, pp. 896–909, 2006.
[40] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.
[41] B. B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," *Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.
[42] R. Singh Rao and M. Gupta, "Design pattern detection by greedy algorithm using inexact graph matching," *International Journal Of Engineering And Computer Science*, vol. 2, no. 10, pp. 3658–3664, 2013.
[43] N. Bozorgvar, A. Rasoolzadegan, and A. Harati, "Probabilistic detection of gof design patterns," *The Journal of Supercomputing*, vol. 79, no. 2, pp. 1654–1682, 2023.
[44] G. Rasool and P. Mäder, "Flexible design pattern detection based on feature types," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 243–252.
[45] N. Nazar, A. Aleti, and Y. Zheng, "Feature-based software design pattern detection," *Journal of Systems and Software*, vol. 185, p. 111179, 2022.
[46] R. Barbudo, A. Ramírez, F. Servant, and J. R. Romero, "Geml: A grammar-based evolutionary machine learning approach for design-pattern detection," *Journal of Systems and Software*, vol. 175, p. 110919, 2021.